10. Functions (Part-2)

10.1 Overloaded functions

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

```
10
2.5
```

In this example, there are two functions called operate, but one of them has two parameters of type int, while the other has them of type double. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two int arguments, it calls to the function that has two int parameters, and if it is called with two doubles, it calls the one with two doubles.

In this example, both functions have quite different behaviors, the int version multiplies its arguments, while the double version divides them. This is generally not a good idea. Two functions with the same name are generally expected to have -at least- a similar behavior, but this example demonstrates that is entirely possible for them not to. Two overloaded functions (i.e., two functions with the same name) have entirely different definitions; they are, for all purposes, different functions, that only happen to have the same name.

Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

10.2 Function templates

Overloaded functions may have the same definition. For example:

```
// overloaded functions
#include <iostream>
using namespace std;

int sum (int a, int b)
{
  return a+b;
}

double sum (double a, double b)
{
  return a+b;
}

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

```
30
2.5
```

Here, sum is overloaded with different parameter types, but with the exact same body.

The function sum could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with generic types, known as function templates. Defining a function template follows the same syntax as a regular function, except that it is preceded by the template keyword and a series of template parameters enclosed in angle-brackets <>:

```
template <template-parameters> function-declaration
```

The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the class or typename keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic sum function could be defined as:

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

It makes no difference whether the generic type is specified with keyword class or keyword typename in the template argument list (they are 100% synonyms in template declarations).

In the code above, declaring SomeType (a generic type within the template parameters enclosed in angle-brackets) allows SomeType to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the function template, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:

name <template-arguments> (function-arguments)

For example, the sum function template defined above can be called with:

```
x = sum<int>(10,20);
```

The function **sum<int>** is just one of the possible instantiations of function template sum. In this case, by using int as template argument in the call, the compiler automatically instantiates a version of sum where each occurrence of SomeType is replaced by int, as if it was defined as:

```
int sum (int a, int b)
{
  return a+b;
}
```

Let's see an actual example:

```
// function template
#include <iostream>
using namespace std;
```

```
11
2.5
```

```
template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

In this case, we have used T as the template parameter name, instead of SomeType. It makes no difference, and T is actually a quite common template parameter name for generic types.

In the example above, we used the function template sum twice. The first time with arguments of type int, and the second one with arguments of type double. The compiler has instantiated and then called each time the appropriate version of the function.

Note also how T is also used to declare a local variable of that (generic) type within sum:

```
T result;
```

Therefore, result will be a variable of the same type as the parameters a and b, and as the type returned by the function.

In this specific case where the generic type T is used as a parameter for sum, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets. Therefore, instead of explicitly specifying the template arguments with:

```
k = sum<int> (i,j);
h = sum<double> (f,g);
```

It is possible to instead simply write:

```
k = sum (i,j);
h = sum (f,g);
```

without the type enclosed in angle brackets. Naturally, for that, the type shall be unambiguous. If sum is called with arguments of different types, the compiler may not be able to deduce the type of T automatically.

Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

```
x and y are equal
```

Note that this example uses automatic template parameter deduction in the call to are_equal:

```
are_equal(10,10.0)
```

Is equivalent to:

```
are_equal<int,double>(10,10.0)
```

There is no ambiguity possible because numerical literals are always of a specific type: Unless otherwise specified with a suffix, integer literals always produce values of type int, and floating-point literals always produce values of type double. Therefore 10 has always type int and 10.0 has always type double.

10.3 Non-type template arguments

The template parameters can not only include types introduced by `class` or `typename`, but can also include expressions of a particular type:

```
// template arguments
#include <iostream>
```

```
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

```
20
30
```

The second argument of the fixed_multiply function template is of type int. It just looks like a regular function parameter, and can actually be used just like one.

But there exists a major difference: the value of template parameters is determined on compile-time to generate a different instantiation of the function fixed_multiply, and thus the value of that argument is never passed during runtime: The two calls to fixed_multiply in main essentially call two versions of the function: one that always multiplies by two, and one that always multiplies by three. For that same reason, the second template argument needs to be a constant expression (it cannot be passed a variable).

10.4 Scopes

Named entities, such as variables, functions, and compound types need to be declared before being used in C++. The point in the program where this declaration happens influences its visibility:

An entity declared outside any block has global scope, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function or a selective statement, has block scope, and is only visible within the specific block in which it is declared, but not outside it.

Variables with block scope are known as local variables.

For example, a variable declared in the body of a function is a local variable that extends until the end of the the function (i.e., until the brace } that closes the function definition), but not outside it:

```
int foo;         // global variable

int some_function ()
{
  int bar;       // local variable
  bar = 0;
}

int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from this function
}
```

In each scope, a name can only represent one entity. For example, there cannot be two variables with the same name in the same scope:

```
int some_function ()
{
  int x;
  x = 0;
  double x;   // wrong: name already used in this scope
  x = 0.0;
}
```

The visibility of an entity with block scope extends until the end of the block, including inner blocks. Nevertheless, an inner block, because it is a different block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside. While outside it, it will still refer to the original entity. For example:

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner scope.
    x = 50;  // sets value to inner x
    y = 50;  // sets value to (outer) y
```

```
inner block:
x: 50
y: 50
outer block:
x: 10
y: 50
```

```
      cout << "inner block:\n";
      cout << "x: " << x << '\n';
      cout << "y: " << y << '\n';
    }
    cout << "outer block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
    return 0;
  }
```

Note that y is not hidden in the inner block, and thus accessing y still accesses the outer variable.

Variables declared in declarations that introduce a block, such as function parameters and variables declared in loops and conditions (such as those declared on a for or an if) are local to the block they introduce.

10.5 Namespaces

Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc...

But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.

The syntax to declare a namespaces is:

```
namespace identifier
{
  named_entities
}
```

Where identifier is any valid identifier and named_entities is the set of variables, types and functions that are included within the namespace. For example:

```
namespace myNamespace
{
  int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace.

These variables can be accessed from within their namespace normally, with their identifier (either a or b), but if accessed from outside the myNamespace namespace they have to be properly qualified with the scope operator ::. For example, to access the previous variables from outside myNamespace they should be qualified like:

```
myNamespace::a
myNamespace::b
```

Namespaces are particularly useful to avoid name collisions. For example:

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the inline specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

```
// namespaces
#include <iostream>
using namespace std;

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

```
5
6.2832
3.1416
```

In this case, there are two functions with the same name: value. One is defined within the namespace foo, and the other one in bar. No redefinition errors happen thanks to namespaces.

Notice also how pi is accessed in an unqualified manner from within namespace bar (just as pi), while it is again accessed in main, but here it needs to be qualified as bar::pi.

Namespaces can be split: Two segments of a code can be declared in the same namespace:

```
namespace foo { int a; }
namespace bar { int b; }
namespace foo { int c; }
```

This declares three variables: a and c are in namespace foo, while b is in namespace bar. Namespaces can even extend across different translation units (i.e., across different files of source code).

10.6 using

The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

```
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

```
5
2.7183
10
3.1416
```

Notice how in main, the variable x (without any name qualifier) refers to first::x, whereas y refers to second::y, just as specified by the using declarations. The variables first::y and second::x can still be accessed, but require fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

```
// using
#include <iostream>
using namespace std;

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

```
5
10
3.1416
2.7183
```

In this case, by declaring that we were using namespace first, all direct uses of x and y without name qualifiers were also looked up in namespace first.

using and using namespace have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

```
// using namespace example
#include <iostream>
using namespace std;
```

```
5
3.1416
```

```
namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

10.7 Namespace aliasing

Existing namespaces can be aliased with new names, with the following syntax:

```
namespace new_name = current_name;
```

10.8 The std namespace

All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the std namespace. Most examples in these tutorials, in fact, include the following line:

```
using namespace std;
```

This introduces direct visibility of all the names of the std namespace into the code. This is done in these tutorials to facilitate comprehension and shorten the length of the examples, but many programmers prefer to qualify each of the elements of the standard library used in their programs. For example, instead of:

```
cout << "Hello world!";
```

It is common to instead see:

```
std::cout << "Hello world!";
```

Whether the elements in the std namespace are introduced with using declarations or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

10.9 Storage classes

The storage for variables with global or namespace scope is allocated for the entire duration of the program. This is known as static storage, and it contrasts with the storage for local variables (those declared within a block). These use what is known as automatic storage. The storage for local variables is only available during the block in which they are declared; after that, that same storage may be used for a local variable of some other function, or used otherwise.

But there is another substantial difference between variables with static storage and variables with automatic storage:

- Variables with static storage (such as global variables) that are not explicitly initialized are automatically initialized to zeroes.
- Variables with automatic storage (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

For example:

```
// static vs automatic storage
#include <iostream>
using namespace std;

int x;

int main ()
{
  int y;
  cout << x << '\n';
  cout << y << '\n';
  return 0;
}
```

```
0
4285838
```

The actual output may vary, but only the value of x is guaranteed to be zero. y can actually contain just about any value (including zero).